## **1. What is Docker?**

*Docker is an open-source containerization platform that allows developers to package applications and their dependencies into containers.  
 These containers can run consistently across different environments — development, testing, and production.*

**2. Why Use Docker?**

### **✅ Benefits of Docker:**

* Consistency: Works across different environments.
* Portability: Containers can run anywhere (local, cloud, etc.).
* Efficiency: Uses less resources than virtual machines.
* Speed: Faster app startup and deployment.

### **🚀 Applying the Solution with Docker**

#### **A Use Case–Driven Problem:**

You are deploying a web application across multiple environments. Manually installing dependencies leads to errors and inconsistencies.

Docker solves this by packaging everything into one portable container image, ensuring it runs the same everywhere.

**3. Docker Installation**

### **🔧 Install Docker on Amazon Linux (EC2)**

After connecting to the EC2 instance, run the following commands:(Restart the session after adding user to the Docker group)

sudo yum update -y

sudo yum install docker -y

sudo service docker start

sudo usermod -aG docker ec2-user

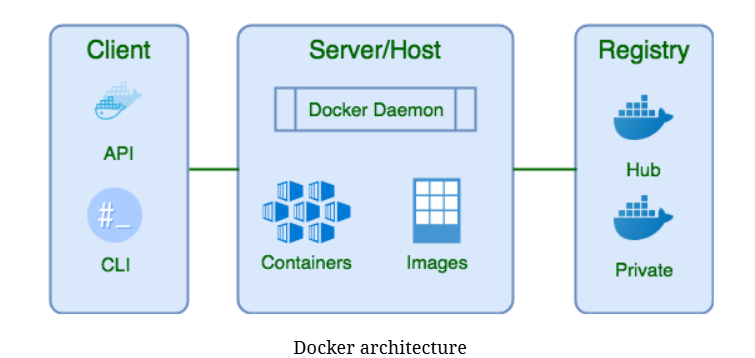
### **▶️ Start Docker Service**

**sudo service docker start**

### **⏹️ Stop Docker Service**

**sudo service docker stop**

## **4. Docker Architecture**



### **1. Docker Client (docker)**

* The CLI tool used to interact with Docker.
* Sends commands to the Docker daemon via REST API.
* Acts as the frontend of Docker.

**📌 Example:**

**docker run nginx**

This command is sent from the client to the daemon to start a container**.**

### **2. Docker Daemon (dockerd)**

* Runs in the background.
* Handles:  
  + Building, running, and managing containers.
  + Networking, storage, and image management.
  + API requests from Docker Client.
  + Communication with other daemons (Swarm mode).

**3.Docker Registry**

* A Docker Registry is a storage and distribution system for Docker images. It allows you to store, share, and retrieve container images.

## 

## **5. Docker Basic Commands**

|  |  |
| --- | --- |
| **Command** | **Description** |
| **docker info** | **Show system-wide Docker information** |
| **docker -v** | **Display Docker version** |
| **docker help** | **List all Docker commands** |
| **docker login** | **Log into Docker Hub** |
| **docker images** | **List all local images** |
| **docker ps** | **List running containers** |
| **docker pull <image>** | **Download image from Docker Hub** |
| **docker run <image>** | **Run a container from an image** |
| **docker build -t <name>** | **Build an image from a Dockerfile** |
| **docker tag <img> <repo>:<tag>** | **Tag an image for pushing** |
| **docker push <repo>:<tag>** | **Push image to a registry** |
| **docker rmi <image>** | **Remove an image** |
| **docker inspect <container>** | **Show container/image details** |

## **6. Writing a Dockerfile**

A Dockerfile is a text document that contains instructions to build a Docker image.

## **Dockerfile Commands :**

|  |  |  |
| --- | --- | --- |
| **Command** | **Description** | **Example** |
| **FROM** | **Sets the base image** | **FROM ubuntu:20.04** |
| **LABEL** | **Adds metadata to the image** | **LABEL maintainer="you@example.com"** |
| **RUN** | **Executes a command while building the image** | **RUN apt-get update && apt-get install -y nginx** |
| **CMD** | **Sets default command to run when container starts** | **CMD ["nginx", "-g", "daemon off;"]** |
| **ENTRYPOINT** | **Sets a fixed command (overrides CMD)** | **ENTRYPOINT ["python3", "app.py"]** |
| **COPY** | **Copies files from host to image** | **COPY . /app** |
| **ADD** | **Like COPY, but can also extract .tar files or use URLs** | **ADD myapp.tar.gz /app** |
| **WORKDIR** | **Sets the working directory for commands** | **WORKDIR /app** |
| **ENV** | **Sets environment variables** | **ENV APP\_ENV=production** |
| **EXPOSE** | **Documents the port the container listens on** | **EXPOSE 80** |
| **VOLUME** | **Creates a mount point inside the image** | **VOLUME /data** |
| **ARG** | **Defines build-time variables** | **ARG APP\_VERSION=1.0** |
| **USER** | **Sets the user for running commands** | **USER appuser** |
| **HEALTHCHECK** | **Defines how to test container health** | **`HEALTHCHECK CMD curl --fail** [**http://localhost:80**](http://localhost:80) |

### **🧱 Sample Dockerfile: Link:** [Simple Docker practice steps](https://docs.google.com/document/d/1iUIIoZtl86kqOX4pB8VKomkYRNE9Vsv5/edit?usp=sharing&ouid=111795996556831533025&rtpof=true&sd=true)

### **7. Docker Volumes:**

**Docker volumes** are a way to persist data in Docker containers. By default, any data written inside a container is lost when the container is removed. Volumes solve this by storing data **outside the container's filesystem**, managed by Docker itself.

### **🔸 Why Use Volumes?**

* Data **persists** after the container is deleted.
* **Share data** between multiple containers.
* Better **performance** than using bind mounts, especially on Linux.
* Docker manages volumes' **location and permissions**.

## **Types of Docker Volumes**

There are **3 main types**:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Description** | **Data Location** | **Managed By** |
| **Volumes** | Default type created with docker volume create | Inside Docker (/var/lib/docker/volumes) | Docker |
| **Bind Mounts** | Mount a specific host path into the container | Anywhere on host (e.g., /home/user/data) | User |
| **tmpfs** | Stores data in memory (RAM) only | RAM | Docker |

**1. Docker Volume (Default / Named Volume)**

### **✅ Best for: Storing data managed by Docker (e.g., database data)**

### **🔧 Example:**

**docker volume create mydata**

**docker run -d --name mysql -v mydata:/var/lib/mysql**

**-e MYSQL\_ROOT\_PASSWORD=root \mysql:5.7**

This stores MySQL data in a volume named mydata.

## **2. Bind Mount**

### **✅ Best for: Sharing code between host and container (e.g., during development)**

### **🔧 Example:**

**docker run -d --name web -v /home/user/app:/usr/share/nginx/html nginx**

Mounts host folder /home/user/app into container at /usr/share/nginx/html.

## **3. tmpfs Mount**

### **✅ Best for: Temporary, in-memory data (auto-clears when container stops)**

### **🔧 Example:**

**docker run -d --name test --tmpfs /app/tmp nginx**

Mounts /app/tmp inside the container as a RAM-based filesystem.

|  |  |  |
| --- | --- | --- |
| **Command** | **Description** | **Example** |
| **docker volume create <name>** | **Creates a named volume** | **docker volume create myvolume** |
| **docker volume ls** | **Lists all volumes** | **docker volume ls** |
| **docker volume inspect <name>** | **Shows detailed info about a volume** | **docker volume inspect myvolume** |
| **docker volume rm <name>** | **Removes a specific volume** | **docker volume rm myvolume** |
| **docker volume prune** | **Removes all unused (dangling) volumes** | **docker volume prune** |
| **docker run -v <volume>:/path/in/container <image>** | **Mounts a volume into a container** | **docker run -v myvolume:/app/data nginx** |
| **docker inspect <container>** | **Shows container info, including mounted volumes** | **docker inspect mycontainer** |
| **docker run -v /host/path:/container/path <image>** | **Bind mount: mount host directory** | **docker run -v /home/user:/data alpine** |
| **docker run --tmpfs /container/path <image>** | **Creates a temporary in-memory volume (tmpfs)** | **docker run --tmpfs /tmp/data alpine** |

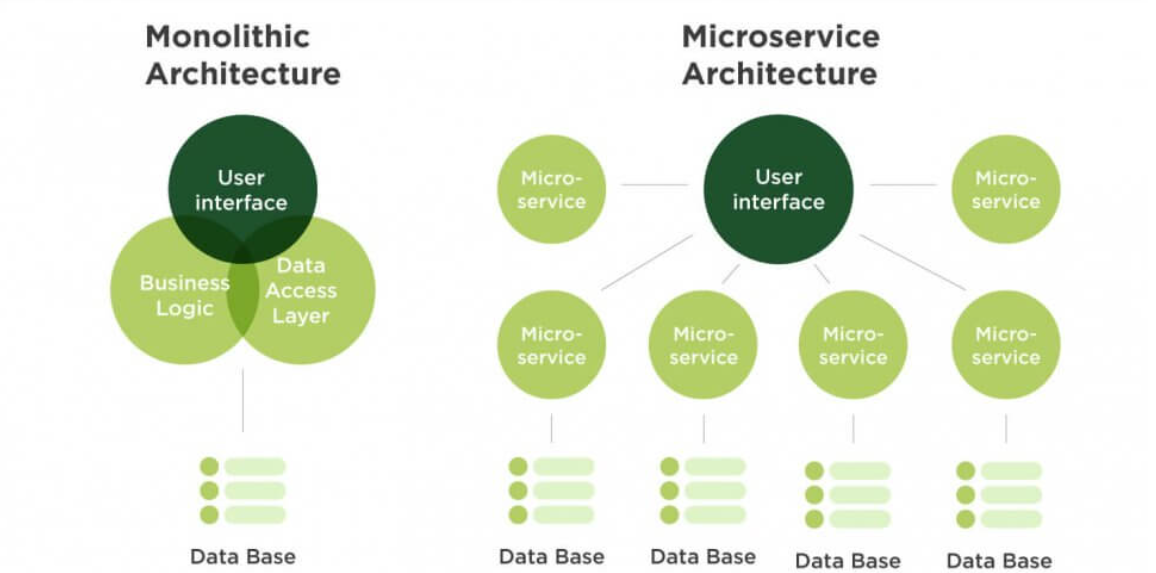
**8. Monolith vs Microservices:**

**What is Monolithic Architecture?**

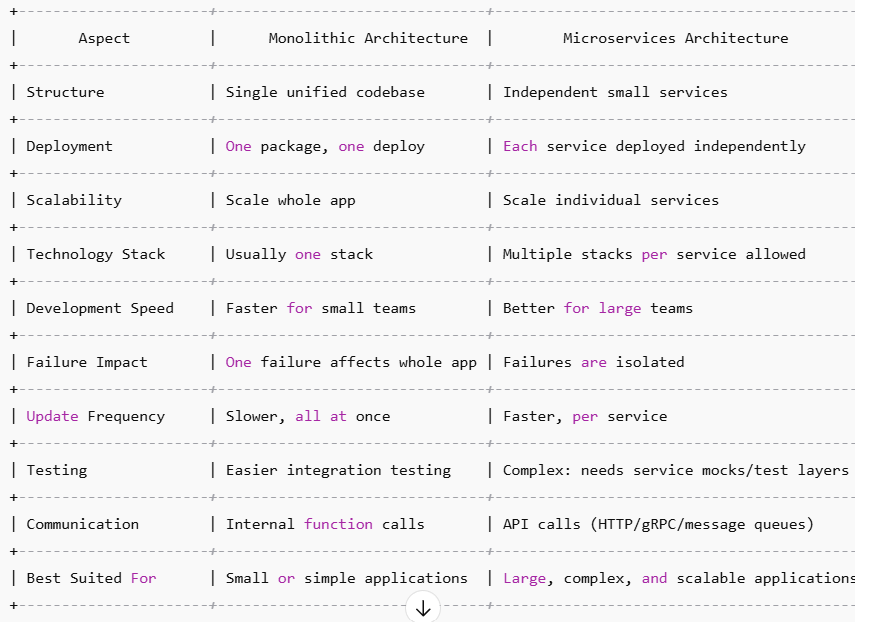
A single-tiered application where all components (UI, business logic, database access) are tightly integrated and run as a single service.

**What is Microservices Architecture?**

An architecture where the application is broken into smaller, independent services, each responsible for a specific business function and running in its own process.



**Comparison table:**

****

## **9) Docker Networking:**

## **Docker Networking allows containers to communicate with each other and with the outside world in an isolated and secure manner. When you create containers, they can be connected to user-defined networks so they talk by name instead of IP addresses.**

## **💡 Why Use Docker Networks?**

## Containers in different projects can be isolated from each other.

## Services can communicate securely without exposing all ports to the host.

## Built-in DNS lets containers resolve each other by service names.

## **✅ Advantages**

## **Service Discovery –** Containers can talk using names like mysql-db instead of IPs.

## **Isolation & Security –** Containers not on the same network cannot access each other.

## **Flexibility –** Easily connect or disconnect containers from networks.

## **Multiple Networks –** You can attach containers to multiple networks for complex architectures.

## **Built-in DNS –** No need for manual IP management.

## **Bridge Between Host & Containers –** Allows selective port exposure to host.

## **📋 Docker Network Commands Table**

|  |  |
| --- | --- |
| **Command** | **Description** |
| **docker network ls** | **List all networks.** |
| **docker network inspect <network>** | **Show details of a network.** |
| **docker network create <name>** | **Create a new user-defined network.** |
| **docker network rm <name>** | **Remove a network.** |
| **docker network prune** | **Remove all unused networks.** |
| **docker network connect <network> <container>** | **Connect an existing container to a network.** |
| **docker network disconnect <network> <container>** | **Disconnect a container from a network** |
| **docker run --network <name>** | **Start a container attached to a specific network.** |

## 

## **📌 Docker Network Types (by default)**

|  |  |  |  |
| --- | --- | --- | --- |
| Network Type | Driver Used | Description | Use Case |
| bridge | bridge | Default for standalone containers. Containers on the same bridge can talk via IP or name. | Running multiple containers on a single host. |
| host | host | Removes isolation between container and host network. Container uses host’s IP and ports directly. | High-performance apps that need direct host network access. |
| none | null | No network at all. Completely isolated container (except manual connections). | Secure or offline processing containers. |

## 

## **10) Install Docker & Docker Compose**

Docker Compose comes bundled with Docker Desktop on Windows & Mac, and with Docker CLI on modern Linux.

### **Windows / Mac**

1. Download & install Docker Desktop.
2. During installation, ensure "Install Docker Compose" is checked.

### **Linux (Ubuntu/Debian example)**

1. **Install Docker:**

**sudo apt update**

**sudo apt install docker.io -y**

**sudo systemctl enable docker --now2.**

1. **Install Docker Compose plugin:  
     
   sudo apt install docker-compose-plugin -y**
2. **Verify:  
     
   docker --version**

**docker compose version**

**11 ) Docker Compose – Meaning & Advantages**

**Docker Compose is a tool that allows you to define and manage multi-container Docker** applications using a simple YAML file (docker-compose.yml).  
 Instead of running multiple docker run commands, you can describe your services (containers), networks, and volumes in one file and start them all together with a single command:

**docker-compose up**

For example, if you have:

* A PHP application container
* A MySQL database container
* A phpMyAdmin container

You can define all three in a single file and run them with one command.

### **Advantages**

1. **Simplified Multi-Container Management**
   * Manage multiple containers as one application with a single command (up or down).
2. **Easy Configuration with YAML**
   * Everything (services, networks, volumes) is declared in a readable docker-compose.yml.
3. **Reusable & Portable**
   * The same file works across environments (dev, staging, production) with minimal changes**.**
4. **Built-in Networking**
   * Containers can communicate using service names (e.g., db for database) without manual IP setup.
5. **Environment Variable Support**
   * Use .env files to manage secrets, credentials, or configuration without hardcoding.
6. **Volume Persistence**
   * Keeps data safe even if containers are removed (via volumes).
7. **Easier Scaling**
   * Quickly scale services (e.g., docker-compose up --scale web=3 to run 3 web containers).
8. **Better Collaboration**
   * Developers can share one YAML file, and everyone can run the exact same setup.

**Here’s a table of common Docker Compose commands with descriptions:**

|  |  |
| --- | --- |
| **Command** | **Description** |
| **docker-compose up** | **Builds (if needed) and starts all containers defined in docker-compose.yml.** |
| **docker-compose up -d** | **Starts containers in detached mode (runs in background).** |
| **docker-compose down** | **Stops and removes containers, networks, and volumes (if not external).** |
| **docker-compose down -v** | **Stops and removes containers and named volumes.** |
| **docker-compose ps** | **Lists all containers managed by the current docker-compose.yml.** |
| **docker-compose logs** | **Shows logs from all containers.** |
| **docker-compose logs -f** | **Follows container logs in real time.** |
| **docker-compose build** | **Builds or rebuilds images for services.** |
| **docker-compose pull** | **Pulls the latest version of images from Docker Hub/registry.** |
| **docker-compose start** | **Starts existing stopped containers (without recreating).** |
| **docker-compose stop** | **Stops running containers without removing them.** |
| **docker-compose restart** | **Restarts containers.** |
| **docker-compose exec <service> <command>** | **Executes a command inside a running container (e.g., docker-compose exec php-app bash).** |
| **docker-compose run <service> <command>** | **Runs a one-off command in a new container.** |
| **docker-compose config** | **Validates and shows the merged configuration.** |
| **docker-compose rm** | **Removes stopped containers.** |

## 

## 

## 

## 

## 

## 

## 

## **1️⃣ What is Docker Swarm?**

Docker Swarm is **Docker’s native container orchestration tool**.  
 It lets you run multiple containers across multiple servers (nodes) in a **cluster** as if they were one big system.

* **Nodes** → Machines (VMs or physical) in the cluster.
* **Manager Node** → Controls and schedules containers.
* **Worker Nodes** → Run the containers.
* **Services** → A definition of tasks (containers) to run.
* **Tasks** → The actual running containers created by a service.

Think of it as **“multi-host Docker”** with built-in scaling, load balancing, and high availability.

## **2️⃣ Advantages of Docker Swarm**

|  |  |
| --- | --- |
| **Feature** | **Benefit** |
| **Built into Docker** | No extra install — comes with Docker Engine. |
| **Easy setup** | Simple commands to init and join nodes. |
| **Scaling** | Scale services up/down in seconds. |
| **Load balancing** | Distributes requests across replicas automatically. |
| **High availability** | If a container fails, Swarm restarts it elsewhere. |
| **Rolling updates** | Update containers with zero downtime. |
| **Service discovery** | Use service names as DNS hostnames (no IP hardcoding). |
| **Overlay networks** | Containers across multiple nodes can communicate. |
| **Secure by default** | Uses mutual TLS between nodes. |

## **3️⃣ Important Docker Swarm Commands**

Here’s a cheat sheet you can keep:

|  |  |
| --- | --- |
| **Command** | **Description** |
| docker swarm init | Initialize a new Swarm cluster on the current node (becomes Manager). |
| docker swarm join --token <token> <manager-ip>:2377 | Join a node to the Swarm as worker/manager. |
| docker node ls | List all nodes in the Swarm cluster. |
| docker service create --name <name> <image> | Create and run a new service. |
| docker service ls | List all running services. |
| docker service ps <service> | See which nodes are running the service's tasks. |
| docker service scale <service>=<replicas> | Scale the service to the given number of replicas. |
| docker service rm <service> | Remove a service. |
| docker stack deploy -c <compose-file> <stack-name> | Deploy a stack from a Compose file. |
| docker stack ls | List stacks in Swarm. |
| docker stack services <stack> | List services in a stack. |
| docker stack ps <stack> | List running tasks in a stack. |
| docker stack rm <stack> | Remove a stack. |

## **4️⃣ How Swarm Networking Works**

* Swarm creates an **overlay network**.
* Services inside the same overlay can talk using **service names** (DNS-based).
* Load balancing is automatic — hitting any node IP on the service port routes to a replica.

**Here are some common Docker troubleshooting tips you can use or even include in your LinkedIn showcase:**

### **1. Container Not Starting**

**Check logs:  
  
docker logs <container\_name>**

* Look for missing environment variables, port conflicts, or image issues.

### **2. Port Already in Use**

**Identify which process is using the port:  
  
sudo lsof -i :<port>**

Stop the conflicting process or change the container port mapping.

### **3. Changes in Code Not Reflecting**

* If using docker-compose, ensure volumes are mounted correctly.

**Rebuild the image:  
  
docker-compose build --no-cache**

### 

### **4. Container Exits Immediately**

* Often caused by the main process finishing. Keep it running by:  
  + Using a foreground process.
  + Adding tail -f /dev/null for debugging.

### **5. Permission Denied Errors**

* Check file permissions when mounting volumes.  
  Use the correct user inside the container:  
  **USER <username>**

### **6. Docker Daemon Not Running**

Restart Docker service: **sudo systemctl restart docker**

### **7. Network Issues Between Containers**

Ensure they are in the same network:  
 **docker network ls**

**docker network inspect <network\_name>**

* In Docker Compose, use service names for host resolution.